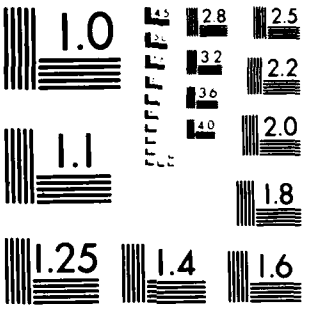
AD-A135 794 THE EXTENSION AND APPLICATION OF GLOBAL COMPACTION TECHNIQUES FOR HORIZON..(U) YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE J A FISHER 08 NOV 83 1/1

UNCLASSIFIED ARO-18483.5-EL DAAG29-81-K-0171 F/G 9/2 NL



Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ARO 18483.5-EL

(12)

AD-A135794

# REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

| | |
|---|---|
| 1. REPORT NUMBER | 2. GOVT ACCESSION NO. A135794 / 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle) The Extension and Application of Global Compaction Techniques for Horizontal Scientific Code | 5. TYPE OF REPORT & PERIOD COVERED FINAL - 9/1/81 to 8/31/83 |
| | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s) Joseph A. Fisher | 8. CONTRACT OR GRANT NUMBER(s) DAAG 29 81 K 0171 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS Yale University, Department of Computer Science, New Haven, CT 06520 | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS P-18483EL |
| 11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office, Post Office Box 12211, Research Triangle Park, NC 27709 | 12. REPORT DATE 11/8/83 / 13. NUMBER OF PAGES 3 |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) | 15. SECURITY CLASS. (of this report) Unclassified / 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

ELECTE DEC 14 1983

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

AP164-GENERATOR, TRACE SCHEDULING, ELI

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report summarizes the work completed in applying trace scheduling to the Floating Point Systems AP 164 array processor. We are involved in an ongoing project focused upon trace scheduling. The two main aspects of the project are the development of a trace scheduling compiler and to design a single instruction multiple data (SIMD) floating point machine that is optimized for trace scheduling. The aspect of this project which has been done under this ARO grant was the development of a trace scheduler for the AP164 attached processor. This

DTIC FILE COPY

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20.
architecture is the most common example of a SIMD machine for which the user writes code. The AP164 offers limited parallelism in that it has three alu's and a limited cross-bar.

# The Extension and Application of Global Compaction Techniques for Horizontal Scientific Code

Final Report

Joseph A. Fisher

November 21, 1983

U. S. Army Research Office

DAAG 2981 K 0171

Yale University



A-1

# Status of the AP164-Generator

This report summarizes the work completed in applying trace scheduling to the Floating Point Systems AP164 array processor. We are involved in an ongoing project focused upon trace scheduling. The two main aspects of the project are the development of a trace scheduling compiler and to design a single instruction multiple data (SIMD) floating point machine that is optimized for trace scheduling. The aspect of this project which has been done under this ARO grant was the development of a trace scheduler for the AP164 attached processor. This architecture is the most common example of a SIMD machine for which the user writes code. The AP164 offers limited parallelism in that it has three alu's and a limited cross-bar.

The compiler can now compile several of the test codes used by the package. This includes most arithmetic code involving conditionals, scalars, vectors, and loops. For simple loops it is competitive with the FPS compiler, which has recently been improved for tight loops. For code involving nested loops preliminary evaluation indicates that it performs better. Work on the code generator and the trace scheduling package is proceeding, although ARO in not involved in the funding.

The code generator is a module in the trace scheduling package being developed at Yale. Another code generator is for a highly parallel processor, ELI, being designed in the same project. The ELI is being designed to be highly suitable for the compiler methods of this project. The compiler is separated into co-routines with very clear interfaces. The modules are 1) the bookkeeper, 2) the disambiguator, and 3)the code generator. No changes are made to the first two modules to change the code generator. Furthermore only a handful of functions are needed for each interface.

While the ELI is being designed for a compiler to write the code, the AP164 was designed for code generation by hand. The compilers for it enable it to do large jobs, but not with the efficiency that carefully written apal would give. Furthermore, as it offers only limited parallelism, it is not the best environment for trace scheduling. The original model (AP120) had no compiler for years and then only realitively weak ones, which greatly limited its use. This work will attempt to explore the benefits of trace scheduling for this limited architecture. Sample test programs have been compiled by 1)trace scheduling using an AP164 code generator, 2)basic block scheduling using the same code generator, and 3)using the FPS compiler. The timings from running these with sample data will serve as data for a preliminary evaluation of trace scheduling for the AP164.

Trace scheduling is implemented by the bookkeeper picking the traces, deciding where to rejoin traces, and doing the bookkeeping necessary to keep the program valid. The traces are handed one by one to the code-generator, which then schedules it as straight line code.

Each trace is a sequence of n address instructions with the main trace being the most likely path through the flow graph representing the program. As the important traces extend across basic block boundaries and have only jumps out from them, a high degree of parallelism has been found in them. The job of a code generator within a trace scheduler is to utilize this parallelism and keep the functional units busy. For the two machines being considered (the FPS164 and the ELI), the main emphasis of the code generators is in finding optimal paths to and from functional units.

Our main heuristic is to use a scheduling order based upon a demand driven analysis of the trace's DAG. The demand driven scheduling, starts at the output variables of an ordered DAG. It then sees what has to be scheduled, to schedule the outputs. This typically would involve a call to the scheduler to schedule the needed variables. This recursive calling of the scheduler produces variables when they are needed or consumed.

The main body of work was done on a DEC-20 using a dialect of LISP. Starting with some code in one of several higher level languages, a module in the trace scheduler hands the compiler naddr traces with various bookkeeping information. The generator first forms a symbol table, then using a DAG from the ELI code generator generate a machine level schedule using demand driven techniques. The schedule is then converted into Apal and transferred to the VAX 780 to which the FPS164 is attached. There it is verified by simulation or execution.

The major difference between this generator and the ELI generator, is that the AP164 generator tries many different paths for each naddr operation. On the ELI paths can be very long and the hardware offers a great deal of parallelism and symmetry, so it is expected that the shortest path (in time and distance) will be adequate. The AP164 is such that paths can easily conflict with each other and keeping paths open to the four floating inputs of two alu's will require several options to be explored. Since the paths are short it is hoped that the overhead

will not be too great.

The basic stages the generator performed by the generator are generation of the symbol table and DAG, finding a list of paths by which the naddr can be performed, picking the best paths, binding the best path to the schedule, and extracting the Apal language program from the schedule. The order of naddr scheduling will be based upon Ruttenberg's demand driven ordering of the DAG developed for the ELI.

At the end of this grant period, the following pieces have been built and are working. The symbol table work is done. The paths are generated for all function units, data pads, and scalar pads for arithmetic code. Path picking is done based upon the shortest in time path, and all binding steps are performed. The extraction of Apal is working for this subset of the full FPS164 machine. (Register assignment is done by hand as this is not implemented yet.)

Since the last progress report, the major piece of the machine that was included was the memory. This has allowed the inclusion of procedures to compile loading and storing of vector elements to and from memory. Also register spilling and loading can then be included. Once all of these pieces are working improvements on the live-dead analysis are needed. These can be ported from the ELI code generator. At the conclusion of the granting period, major traces can be compiled successfully for a large body of code. These traces do not need the missing modules, which are the register allocator, a complete live-dead analysis, and a module to start with variables in one set of locations and move them to an arbitrarily different set of final locations.

The interface with the bookkeeper has been implemented and is finished except for the improvements that are needed in the live-dead analysis. Trace scheduling has been tested on some small basic sections of code with good improvements over basic block scheduling. This now working scheduler will be enhanced step by step such that it can deal with the full set of test problems.

As examples of the present compiler, we present examples involving two algorithms. These are the main traces of a dotproduct and a matrix multiply, both not unrolled and unrolled. For the not unrolled dotproduct, 3 cycles per interation of the loop for the FPS compiler, while 27 cycles for the trace scheduler. For unrolling five times, the FPS: 87 cycles, the trace scheduler: 27 cycles. The matrix multiply was 40 cycles with no unrolling for the FPS and 36 cycles with two unrollings for the trace compiler. Thus the dotproduct results were one term in 3 (FPS) or 5.4 (trace) cycles for best cases for each. For the matrix multiply it was FPS:40 and trace:18 cycles for best case. Futhermore for the unrolled dotproduct it was FPS:17.4 vs trace:5.4. Our analysis of these and other results, is that loop folding performs better for the simplest loops but as loop complexity increases the trace scheduling method perfoms better.

At this time it would seem that the main stumbling block to achieving a full compiler in a reasonable amount of time is the irregularity of the architecture of the AP164 and the tendency of that to reflect itself in the code. The architecture doesn't lend itself well to any abstract

model. The machine is a combination of cross bars and busses with many unusual features.

The work is also directed at yielding a module that can be used by the numerical analysis group. Important subroutines hopefully will be compiled using the trace scheduler. These can be written in "tiny lisp" which allows full expression of structured fortran concepts. It will also allow careful expression of the algebra to achieve a minimal critical path. Thus the code can be tuned by the numerical analyst without having to resort to apal. While the FPS compiler can be used for the outer control parts such as command parsing and I/O.

Further it is hoped that insight into the workings of trace scheduling will be of benefit to the development of this technique. For example the present results are consistent with the supposition that for simple loops, trace scheduling will remain competive with software pipelining and loop folding if it unrolls the loops. Futhermore that for nested loops and loops involving complex conditionals, trace scheduling will perform significantly better. Also the interaction of the AP164 architecture and the trace scheduler, should help in sorting what machine features enhance trace scheduling and which are in conflict with it. This information will be of use in designing ELI.

## LIST OF PUBLICATIONS

J. A. Fisher.
COMPUTER SYSTEMS ARCHITECTURE AT YALE The Enormous Longword Instruction (ELI) Machine Progress and Research Plans.
Technical Report 241, Yale University Department of Computer Science, July, 1982.

J. A. Fisher.
Very long instruction word architectures and the ELI-512.
Technical Report 253, Yale University Department of Computer Science, April, 1983.
Appeared in 10th Annual International Architecture Conference, Stockholm, JUNE '83.

John C. Ruttenberg & Joseph A. Fisher.
Lifting the Restriction of Aggregate Data Motion in.
In IEEE International Workshop on Computer Systems Organization, pages 211-215. IEEE, March, 1983.